09/792,357 #4

BUNDESREPUBLIK DEUTSCHLAND

Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

Aktenzeichen:	199 54 407.7
Anmeldetag:	12. November 1999
Anmelder/Inhaber:	GfS Systemtechnik GmbH & Co KG, Aachen/DE
Bezeichnung:	Verfahren zum direkten Aufrufen einer Funktion mittels eines Softwaremoduls durch einen Prozessor mit einer Memory-Management-Unit (MMU)
IPC:	G 06 F 9/46

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.

München, den 21. September 2000
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag

Hiebl

A 9161
06/00

Beschreibung:

Verfahren zum direkten Aufrufen einer Funktion mittels eines Softwaremoduls durch einen Prozessor mit einer Memory-Management-Unit (MMU)

Die Erfindung betrifft ein Verfahren zum direkten Aufrufen einer Zielfunktion mittels einer Startfunktion durch einen Prozessor mit einer Memory-Management-Unit (MMU) in einem durch ein Betriebssystem betriebenen Computer.

Bei Multitasking-Betriebssystemen werden mehrere Programme (quasi-) gleichzeitig ausgeführt. Da Programme nicht immer fehlerfrei ablaufen, müssen Multitasking-Betriebssysteme im Fall eines Programmfehlers den Schaden möglichst gering halten. Dazu isolieren Multitasking-Betriebssysteme den Speicher (engl.: Memory) der einzelnen Programme derart voneinander, daß das Fehlverhalten eines Programmes keines der anderen Programme in Mitleidenschaft ziehen kann. Zur Vereinfachung wird im folgenden von Betriebssystemen gesprochen, wobei stets Multitasking-Betriebssysteme gemeint sind.

Die Memory-Isolierung besteht in der Trennung zwischen virtuellem und physikalischem Memory. Memoryzugriffe von Programmen finden stets auf virtuellem und nicht auf physikalischem Memory statt. Virtuelles Memory wird vom Prozessor durch Auslesen von Tabellen auf physikalisches Memory abgebildet. Dazu verfügt der Prozessor über eine Memory-Management-Unit (kurz: MMU). Es ist Aufgabe der Betriebssysteme, diese Tabellen anzulegen und zu verwalten. Diese Tabellen werden als Memory-Kontexte bezeichnet. Die Memory-Kontexte liegen selbst im Memory des Rechners und werden von der MMU gelesen. Im Gegensatz zu Programmen finden Memoryzugriffe der MMU stets auf physikalischem und nicht auf virtuellem Memory statt. Die MMU benötigt zum Zugriff auf einen Memory-Kontext dessen physikalische Adresse. Dazu verfügt der Prozessor über ein MMU-Steuerregister. Im MMU-Steuerregister ist die physikalische Adresse des aktuell gültigen Memory-Kontextes abgelegt.

Jedem Programm ist ein eigener Memory-Kontext zugeordnet. Ein Programm mit seinem eigenen Memory-Kontext wird als Task bezeichnet. Die Memory-Kontexte werden von Betriebssystemen derart gestaltet, daß sie sich bezüglich des physikalischen Memorys nicht überlappen. Dadurch ist die Memory-Isolierung zwischen den Tasks gewährleistet. Die Fehlfunktion eines Programms findet somit nur in einem Memory-Kontext (Speicherkontext) statt und kann andere Programme in anderen Memory-Kontexten nicht beeinflussen.

Das Prinzip der physikalischen Isolierung der Memory-Kontexte zweier Tasks ist in der Figur 1 der beigefügten Zeichnungen dargestellt.

Durch die Memory-Isolierung wird ein Datenaustausch zwischen Tasks unmöglich gemacht. Der Datenaustausch ist jedoch notwendig, um die Ausgangsdaten einer Task einer anderen Task zur Verfügung zu stellen (z.B. für den Datenaustausch zwischen einem Datenbankprogramm und einem Textverarbeitungsprogramm). Betriebssysteme bieten daher Mittel zur Intertaskkommunikation. Durch die Intertaskkommunikation darf jedoch nicht jene Lücke wieder geöffnet werden, die durch Memory-Isolierung geschlossen wurde. Deswegen erfolgt die Intertaskkommunikation von Betriebssystemen generell durch ein Umkopieren von Daten. Die Betriebssysteme "transportieren" gewissermaßen eine Kopie der Daten von einer Task zur anderen. Durch diesen Mechanismus können Tasks Daten austauschen, ohne gegenseitigen Zugriff auf ihre Daten zu haben.

Der Datenaustausch über ein Betriebssystem hat den Nachteil, daß der höchst mögliche Datendurchsatz wegen des Umkopierens der Daten nicht erreicht werden kann. Es gibt jedoch Anwendungen, in denen es notwendig ist, den höchst möglichen Datendurchsatz zu erreichen. Für solche Anwendungen bieten Betriebssysteme die Verwendung von Shared-Memory (gemeinsamer Speicher) zum direkten gegenseitigen Datenzugriff verschiedener Tasks an. Bei Shared-Memory sind die Memory-Kontexte der beteiligten Tasks derart gestaltet, daß sie sich einen bestimmten physikalischen Memorybereich teilen.

In Fig. 2 der beigefügten Zeichnungen ist das Prinzip des gemeinsamen Speicherzugriffs zweier Tasks dargestellt.

Shared-Memory widerspricht dem Prinzip der Memory-Isolierung. Dieser Widerspruch ist jedoch nicht problematisch, da Tasks nicht beliebig zur Verwendung von Shared-Memory veranlaßt werden können. Wenn Tasks untereinander den höchst möglichen Datendurchsatz erreichen sollen, dann müssen sie Programmfunktionen umfassen, die explizit Shared-Memory anfordern. Eine Task muß daher bewußt durch den Programmierer so konfiguriert werden, daß sie mit bestimmten anderen Tasks Shared-Memory verwendet. Der Programmierer wird daher besondere Rücksicht auf die Gefahren durch Fehlfunktionen der anderen Tasks nehmen, um weitere Auswirkungen derartiger Fehlfunktionen möglichst zu vermeiden.

Neben dem Datenaustausch ist auch der Ereignisaustausch, d.h. der Austausch des Zustandswechsels von Daten, in der Regel nur über die Intertaskkommunikation der Betriebssysteme vorgesehen. Um nicht wieder die Lücke zu öffnen, die durch die Memory-Isolierung geschlossen wurde, werden Ereignisse von Betriebssystemen registriert und zu gegebener Zeit weitergeleitet. Die Betriebssysteme "transportieren" Ereignisse mit ähnlichen Mechanismen wie Daten von einer Task zur anderen.

Der Ereignisaustausch über ein Betriebssystem hat den Nachteil, daß der Weiterleitungszeitpunkt von dem Betriebssystem und den anderen durch das Betriebssystem ausgeführten Tasks abhängt. Es ist also nicht möglich, einen Ereignisaustausch mit optimaler Determiniertheit, d.h. mit exakter Bestimmtheit des Zeitpunktes der Weiterleitung, durchzuführen. Es gibt jedoch eine Anzahl von Anwendungen, bei denen es notwendig ist, die höchst mögliche Determiniertheit zu erreichen. In Analogie zu Shared-Memory läge es nun nahe, daß die Betriebssysteme einen entsprechenden Bypass-Mechanismus anbieten. Keines der existierenden Betriebssysteme bietet jedoch einen solchen Mechanismus.

Es ist daher die Aufgabe der vorliegenden Erfindung, ein Verfahren zum direkten und unmittelbaren Ereignisaustausch zwischen verschiedenen Tasks zu schaffen.

Diese Aufgabe wird dadurch gelöst, daß die Startfunktion Bestandteil einer ersten Task mit einem ersten Speicherkontext ist und die Zielfunktion in

einem anderen Speicherkontext liegt und daß die erste Task einen Kontextwechsel von dem ersten in den anderen Speicherkontext durchführt, der nach der Ausführung der Zielfunktion rückgängig gemacht wird.

Mit anderen Worten umfaßt die erste Task eine Funktion, die unabhängig vom Betriebssystem einen Aufruf der Zielfunktion steuert und durchführt. Die Zielfunktion wird somit unter Umgehung des Taskwechslers (engl.: Scheduler) des Betriebssystems unmittelbar und mit höchster Determiniertheit aufgerufen. Die erste Task übernimmt dabei eine der typischen Aufgaben des Betriebssystems.

Die bevorzugten Verfahrensschritte zur optimalen und betriebssicheren Durchführung dieses Verfahrens ergeben sich aus den Unteransprüchen und der nachfolgenden Beschreibung von Ausführungsformen der Erfindung. Die Erfindung bezieht sich ferner auf ein Softwareprogramm zur Durchführung eines erfindungsgemäßen Verfahrens sowie auf einen maschinenlesbaren Datenträger, auf dem ein derartiges Softwareprogramm abgespeichert ist.

Softwaretechnisch geschieht die direkteste (und determinierteste) Form der Ereignisweiterleitung durch einen Funktionsaufruf. Daher wird in einer Task eine Funktion einer anderen Task aufgerufen, was dem Aufrufen einer Funktion in einem anderen Memory-Kontext entspricht. Ein solcher Aufruf findet im Gegensatz zur Intertaskkommunikation „direkt" statt und wird daher als Direktaufruf bezeichnet.

Der Direktaufruf ist ein Dienst, der von einer Task angeboten wird. Die Task, die den Dienst anbietet, eine Funktion einer anderen Task direkt aufzurufen, wird daher als Server bezeichnet. Der Programmteil des Servers, der den Direktaufruf durchführt, wird als Startfunktion bezeichnet. Die zweite Task, von der eine Funktion direkt aufgerufen werden soll, wird als Client bezeichnet, da sie den Dienst des Servers in Anspruch nimmt. Die direkt aufzurufende Funktion innerhalb des Clients wird als Zielfunktion bezeichnet.

Ähnlich der Verwendung von Shared-Memory muß der erfindungsgemäße Funktionsaufruf durch den Programmierer in dem Client vorgesehen werden. Der direkte Aufruf einer Funktion einer Task kann also nicht durch andere

Programme erzwungen werden. Client und Server müssen bewußt aufeinander abgestimmt werden, um die höchst mögliche Determiniertheit mittels Direktaufruf zu erreichen.

Um die Allgemeingültigkeit des Verfahrens zu gewährleisten, muß die Zielfunktion auch vom Client selbst aufrufbar bleiben. Weiterhin sollten keine compilerspezifischen Einstellungen oder Funktionen eingesetzt werden, da solche Spezialitäten nicht auf allen gängigen Rechnerplattformen einheitlich verfügbar sind.

Um einen Direktaufruf vorzubereiten, muß der Client die Memory-Adresse (kurz: Adresse) seiner Zielfunktion an den Server übermitteln. Dazu nutzt der Client die Intertaskkommunikation der Betriebssysteme. Der Server hat zur Laufzeit die Aufgabe, den Direktaufruf durchzuführen. Das Zusammenwirken von Client und Server ist in Fig. 3 der Zeichnungen dargestellt.

Die im Speicherkontext (kurz: Kontext) des Clients gültige Adresse der Zielfunktion ist jedoch im Kontext des Servers nicht gültig, da es sich um eine virtuelle Adresse handelt. Damit der Server diese Adresse sinnvoll verwenden kann, muß er die physikalische Adresse der Zielfunktion kennen. Die physikalische Adresse der Zielfunktion ist im Kontext des Clients abgelegt. Zur Ermittlung der physikalischen Adresse der Zielfunktion muß also ein lesender Zugriff auf den Kontext des Clients ermöglicht werden.

Da der Server wegen später aufgeführter Gründe unbedingt den lesenden Zugriff auf den Kontext des Clients benötigt, wird die Adresse des Client-Kontextes an den Server übermittelt. Da der Server auf die virtuelle Adresse des Client-Kontextes nicht zugreifen kann, muß der Client die physikalische Adresse seines Kontextes an den Server übermitteln.

Im MMU-Steuerregister des Prozessors ist die physikalische Adresse des Kontextes abgelegt. Der Client ermittelt die physikalische Adresse seines Kontextes durch Auslesen des MMU-Steuerregisters.

Der Server kann jedoch nicht lesend auf physikalisches Memory zugreifen, sondern nur auf virtuelles Memory seines eigenen Kontextes. Die Betriebssysteme bieten jedoch Mechanismen an, um physikalisches Memory in einen

Kontext einzublenden. Dieser Vorgang wird als Mapping (deutsche Bedeutung „Abbilden") bezeichnet. Durch Mapping der physikalischen Adresse des Client-Kontextes in den Server-Kontext hätte der Server die Möglichkeit, den Client-Kontext zu lesen. Damit kann er die physikalische Adresse aus der im Client-Kontext gültigen virtuellen Adresse der Zielfunktion ermitteln.

Durch Mapping der physikalischen Adresse der Zielfunktion in den Server-Kontext hätte die Startfunktion die Möglichkeit, den Direktaufruf durchzuführen.

Im allgemeinen greifen Funktionen jedoch auf absolut adressierte Daten zu. Außerdem können Funktionen auch weitere Funktionen aufrufen. Ferner können innerhalb der Funktionen selbst absolut adressierte Sprünge durchgeführt werden. Die Kenntnis der physikalischen Adresse der Zielfunktion genügt daher nur dann, wenn die Zielfunktion nicht auf absolut adressierte Daten zugreift, keine weiteren Funktionen aufruft und keine absolut adressierten Sprünge enthält. Dieser Fall kommt jedoch in der Praxis kaum vor, da eine Funktion mit solchen Einschränkungen fast keine sinnvollen Ergebnisse erzielen kann.

Um diese Einschränkungen aufzuheben, könnte von allen absoluten virtuellen Adressen in der Zielfunktion die physikalische Adresse ermittelt und in den Server-Kontext gemappt werden. Dann müßten in der Zielfunktion alle im Client-Kontext gültigen Adressen in die entsprechenden gemappten Adressen im Server-Kontext ersetzt werden. Dieses Verfahren wäre jedoch sehr aufwendig, da der Programmcode dazu disassembliert werden muß. Außerdem würde ein Verändern der Adressen in der Zielfunktion bedeuten, daß die Zielfunktion (und alle Funktionen, die von ihr aufgerufen werden) im Client-Kontext selbst nicht mehr lauffähig wäre. Von dieser Verfahrensvariante (Verfahrensvariante 1) wird daher Abstand genommen.

Statt dessen wird gemäß dem Patentanspruch 1 ein Verfahren beschrieben, in dem die Startfunktion aus ihrem Kontext in den Kontext der Zielfunktion wechselt.

Der Server kennt die physikalische Adresse des Client-Kontextes. Anstelle des Mappings der Zielfunktion in seinen eigenen Kontext kann die Startfunktion das MMU-Steuerregister mit der physikalischen Adresse des Client-Kontextes beschreiben. Damit hat die Startfunktion einen Kontextwechsel zum Client durchgeführt. Sie kann den Direktaufruf durchführen, da alle Daten und weiteren Funktionen, die von der Zielfunktion aufgerufen werden, im Kontext des Clients liegen, in dem der Aufruf stattfindet. Nach der Durchführung des Direktaufrufes beschreibt die Startfunktion das MMU-Steuerregister wieder mit dem ursprünglichen Wert und wechselt dadurch zurück in den Server-Kontext.

Damit diese Variante des Direktaufrufes (Verfahrensvariante 2) funktionieren kann, muß die Startfunktion im Shared-Memory von Client und Server liegen. Ansonsten würde im Moment des Wechsels vom Server-Kontext in den Client-Kontext die Startfunktion ihre Gültigkeit verlieren, und eine Rückkehr des Zielfunktionsaufrufes in die Startfunktion wäre nicht mehr möglich. Die Betriebssysteme bieten jedoch keine Möglichkeit, um Programmcode in Shared-Memory zu legen. Nur Daten können im Shared-Memory liegen. Die fehlende Möglichkeit der Betriebssysteme, Programmcode in Shared-Memory zu legen, kann i.A. nachgebildet werden durch schreibende Zugriffe auf den entsprechenden Kontext. Im konkreten Fall würde der Teil des Server-Kontextes, der die Startfunktion enthält, zum Client-Kontext hinzukopiert werden. Durch diesen Kopiervorgang liegt die Startfunktion im Kontext des Clients, so daß die Rückkehr des Zielfunktionsaufrufes in die Startfunktion möglich wird.

Es kommt jedoch ein Problem hinzu, denn ein Kontextwechsel gehört zu den ureigensten Aufgaben der Betriebssysteme. Die Betriebssysteme können daher einen eigenmächtigen Kontextwechsel durch eine Task nicht verarbeiten. Daher speichern die Betriebssysteme beim Taskwechsel (engl.: Scheduling) die physikalischen Kontextadressen der Tasks ab, statt sie jedesmal aus dem MMU-Steuerregister auszulesen. Wenn nun eine Task selbständig einen Kontextwechsel durchgeführt hat und dann vom Taskwechsler (engl.: Scheduler) eines Betriebssystems unterbrochen wird, wird das Betriebssystem der Task bei der nächsten Zuteilung von Rechenzeit ihren ursprünglichen Kontext zuweisen, und nicht den Kontext, in den die Task gewechselt hatte. Damit würde die Zielfunktion im falschen Kontext ausgeführt werden.

Der Server muß daher zur Laufzeit der Startfunktion und der aufgerufenen Zielfunktion ein Scheduling durch die Betriebssysteme unbedingt vermeiden. Diese Bedingung an den Server wird als Serverbedingung bezeichnet.

Weiterhin muß gewährleistet sein, daß von der Zielfunktion keine Betriebssystemfunktion aufgerufen wird, da die Betriebssysteme einen Aufruf aus diesem Kontext nicht verarbeiten können (sie haben ja schließlich nicht in diesen Kontext gewechselt). Das Verhalten der Betriebssysteme wäre bei solchen Aufrufen nicht vorhersehbar. Diese Bedingung an den Client wird als Erste Clientbedingung bezeichnet.

Da der Direktaufruf asynchron zur Laufzeit des Clients stattfindet, und zwar derart, daß während des Direktaufrufes der Client wegen der Serverbedingung unterbrochen ist, kann es zu Problemen kommen, wenn der Client selbst die Zielfunktion (oder Funktionen, die von der Zielfunktion aufgerufen werden) aufruft. Wenn während des Aufrufes der Zielfunktion durch den Client ein Scheduling zum Server stattfindet, der dann seinerseits ebenfalls die Zielfunktion aufrufen würde, dann wäre die Zielfunktion zum zweiten Mal aufgerufen, bevor der erste Aufruf beendet wäre. Ein solcher Zweitaufruf verlangt von einer Funktion die Eigenschaft, wiedereintrittsfähig zu sein. Wiedereintrittsfähige Funktionen sind sehr aufwendig zu konstruieren. Damit diese aufwendige Konstruktion nicht notwendig ist, darf der Client nur dann selbst die Zielfunktion (oder Funktionen, die von der Zielfunktion aufgerufen werden) aufrufen, wenn der Server die Zielfunktion mit Sicherheit nicht aufrufen kann. Diese Bedingung an den Client wird als Zweite Clientbedingung bezeichnet.

Weiterhin darf der Client zu Zeitpunkten, in denen der Server die Zielfunktion aufrufen kann, nur derart auf Daten zugreifen, die durch den Aufruf der Zielfunktion verwendet oder verändert werden könnten, daß der Zugriff innerhalb eines Prozessortaktes abgeschlossen oder über Flaggen verriegelt ist. Andernfalls könnte der Client, während er außerhalb seiner Zielfunktion auf Daten zugreift, mitten im Datenzugriff vom Server unterbrochen werden. Ein Zugriff auf Daten, der innerhalb eines Prozessortaktes abgeschlossen ist, wird als atomarer Zugriff bezeichnet. Diese Bedingung an den Client wird als Dritte Clientbedingung bezeichnet.

Der Server muß zur Erfüllung der Serverbedingung den Scheduler der Betriebssysteme deaktivieren. Der Scheduler der Betriebssysteme arbeitet interruptgesteuert. Die Startfunktion muß daher über ein Prozessorsteuerregister die Interruptverarbeitung anhalten und stellt dadurch sicher, daß sie nicht vom Scheduler eines Betriebssystems unterbrochen werden kann.

Der Client erfüllt die Erste Clientbedingung durch Unterlassung von expliziten Betriebssystemaufrufen in der Zielfunktion. Allerdings muß auch sichergestellt sein, daß in der Zielfunktion keine impliziten Betriebssystemaufrufe stattfinden. Implizite Betriebssystemaufrufe finden statt, wenn Ausnahmesituationen eintreten wie z.B. eine Division durch Null. Daher muß die Zielfunktion fehlerfrei programmiert sein, um keine Ausnahmesituationen zu provozieren, die einen Betriebssystemaufruf zur Folge haben.

Die Betriebssysteme lagern bei Mangel an physikalischem Arbeitsspeicher automatisch Teile des Speichers (Memory) auf Festplatte aus. Greift eine herkömmliche Task auf ausgelagertes Memory zu, dann tritt eine Ausnahmesituation ein, die einen impliziten Betriebssystemaufruf bewirkt. Im Rahmen dieses Betriebssystemaufrufes wird das ausgelagerte Memory von der Festplatte gelesen und in das physikalische Memory (Arbeitsspeicher) geschrieben. Anschließend greift die Task erfolgreich auf das gewünschte Memory zu, ohne den automatischen Betriebssystemaufruf bemerkt zu haben. Ein derartiger implizierter Betriebssystemaufruf während der Laufzeit der direkt aufgerufenen Zielfunktion würde einen kritischen Zustand provozieren, da der Scheduler des Betriebssystems davon ausgeht, daß der Kontext des Servers und nicht der Kontext des Clients aktiviert ist.

Damit solche Ausnahmesituationen ausgeschlossen werden können, muß die Auslagerung des Memorys, auf das die Zielfunktion zugreift, verhindert werden. Die Betriebssysteme bieten Mechanismen an, um die Auslagerung von Memory zu verhindern. Dieser Vorgang wird als Locking (deutsche Bedeutung „Sperren") bezeichnet. Zum Einhalten der Ersten Clientbedingung muß daher das gesamte Memory, das im Zugriff der Zielfunktion liegt, gelockt sein.

Der Client erfüllt die Zweite Clientbedingung durch Unterlassung des Zielfunktionsaufrufes in den Zeitpunkten, in denen die Möglichkeit besteht,

daß der Server die Zielfunktion aufruft. Da der Server zum Aufrufen der Zielfunktion vom Client aktiviert werden muß, kann der Client während einer derartigen Aktivierung den Aufruf der eigenen Zielfunktion blockieren.

Der Client erfüllt die Dritte Clientbedingung durch Beschränkung auf atomare Datenzugriffe oder über eine Verriegelung durch Flaggen beim Zugriff auf Daten, die durch den Aufruf der Zielfunktion verwendet oder verändert werden könnten, wenn die Möglichkeit besteht, daß der Server die Zielfunktion aufruft.

Soweit wären die Bedingungen für den Aufruf der Zielfunktion durch einen Kontextwechsel in der Startfunktion geklärt.

Allerdings gehört das Locken (Sperren) von Programmcode und Daten in seiner notwendigen Gesamtheit nicht zu den Standardaufgaben der Softwareentwicklung. Es ist daher wahrscheinlich, daß dabei Fehler gemacht werden. Es besteht ferner die Schwierigkeit, daß die Betriebssysteme in Abhängigkeit vom Bedarf anderer Tasks das Memory einer bestimmten Task auslagern. Das Auslagern geschieht also zu unvorhersehbaren Zeitpunkten. Da die Betriebssysteme zu nicht vorhersehbaren Zeitpunkten auslagern, ist es nicht testbar, ob alle notwendigen Speicherbereiche gelockt (gesperrt) sind. Dieses Testbarkeitsproblem verlangt nach einer Lösung, die von der Verfahrensvariante 2 nicht geboten werden kann.

Ziel der nun beschriebenen Verfahrensvariante 3 ist, daß nicht gelocktes Memory der Zielfunktion nicht zu unvorhersehbaren Zeitpunkten, sondern unmittelbar beim ersten Aufruf der Zielfunktion erkennbar ist. Dazu wird in der Startfunktion kein Kontextwechsel in den Client-Kontext, sondern in einen neu angelegten Kontext durchgeführt. Das Anlegen eines neuen Kontextes bedeutet eine Konfiguration der MMU des Prozessors. Der neue Kontext wird zusammengesetzt aus Teilkopien des Client-Kontextes und des Server-Kontextes. Der neue Kontext wird daher als kopierter Kontext bezeichnet und bildet den zweiten Kontext, in den aus dem Server-Kontext heraus gewechselt wird. Der Client-Anteil des kopierten Kontextes umfaßt genau diejenigen Teile des Client-Kontextes, die gelocktem Memory ent-

sprechen. Der Server-Anteil des kopierten Kontextes umfaßt die Startfunktion und alle Daten, die sie verwendet.

Die Verfahrensvariante 3 löst das Testbarkeitsproblem der Verfahrensvariante 2 dadurch, daß nicht gelocktes (gesperrtes) Memory auch nicht im kopierten Kontext vorhanden ist. Ein Zugriff der Zielfunktion auf solches Memory macht sich unmittelbar bemerkbar, da er unzulässig ist, wenn die MMU des Prozessors in den kopierten Kontext gewechselt hat.

Im Rahmen der Beschreibung des Verfahrens zum direkten Aufruf einer Funktion in einem anderen Memory-Kontext durch Konfiguration der MMU des Prozessors wurde anhand der Verfahrensvarianten 1 und 2 die grundsätzliche Problematik erläutert. Die Verfahrensvariante 1 ermöglicht nicht den direkten Aufruf einer für die Ausführung sinnvoller Aufgaben ausreichend komplexen Funktion. Die Verfahrensvariante 2 ist für einen derartigen Aufruf geeignet, weist aber insbesondere Probleme bei der Testbarkeit auf.

Die Verfahrensvariante 3, welche einen Kontextwechsel nicht in den Client-Kontext, sondern in einen kopierten Speicherkontext durchführt, wobei das Memory dieses kopierten Speicherkontextes vollständig gesperrt (gelockt) ist, stellt die optimale Ausführungsform der Erfindung dar.

In Analogie zu Shared-Memory als Mittel zum Erreichen des höchst möglichen Datendurchsatzes zwischen Tasks liefert das dargestellte erfindungsgemäße Verfahren ein Mittel zum Erreichen der höchst möglichen Determiniertheit für einen Ereignisaustausch zwischen Tasks. So wie Shared-Memory eine direkte Datenkopplung ermöglicht, so ermöglicht das dargestellte Verfahren eine direkte Ereigniskopplung.

* * * * * * *

Ansprüche:

1. Verfahren zum direkten Aufrufen einer Zielfunktion mittels einer Startfunktion durch einen Prozessor mit einer Memory-Management-Unit (MMU) in einem durch ein Betriebssystem betriebenen Computer, **dadurch gekennzeichnet**, daß die Startfunktion Bestandteil einer ersten Task mit einem ersten Speicherkontext ist und die Zielfunktion in einem anderen Speicherkontext liegt und daß die erste Task einen Kontextwechsel von dem ersten in den anderen Speicherkontext durchführt, der nach der Ausführung der Zielfunktion rückgängig gemacht wird.

2. Verfahren nach Anspruch 1, **dadurch gekennzeichnet**, daß zur Durchführung des Kontextwechsels das MMU-Steuerregister durch die erste Task mit der physikalischen Adresse des Speicherkontextes der Task beschrieben wird, in dem die Zielfunktion enthalten ist.

3. Verfahren nach Anspruch 1, **dadurch gekennzeichnet**, daß Teile der ersten Task sowie der die Zielfunktion enthaltenden zweiten Task in einen neuen Speicherkontext kopiert werden und der Kontextwechsel in diesen neuen Speicherkontext erfolgt.

4. Verfahren nach Anspruch 3, wobei der Computer einen Arbeitsspeicher (RAM) und einen Massenspeicher, z.B. einer Festplatte, aufweist und wobei das Betriebssystem bei Bedarf Teile des Arbeitsspeichers in den Massenspeicher auslagert, **dadurch gekennzeichnet**, daß der Speicherbereich des kopierten Speicherkontextes zumindest während der Laufzeit der durch die Startfunktion aufgerufenen Zielfunktion gegen Auslagerung aus dem Arbeitsspeicher gesperrt ist.

5. Verfahren nach einem der vorangehenden Ansprüche, **dadurch gekennzeichnet**, daß die Startfunktion während der Laufzeit der Zielfunktion einen Taskwechsel durch das Betriebssystem vermeidet, indem sie über ein Prozessorsteuerregister die Interrupt-Verarbeitung deaktiviert.

6. Verfahren nach einem der vorangehenden Ansprüche, **dadurch gekennzeichnet**, daß die Zielfunktion keine Programmschritte umfaßt, die einen Aufruf des Betriebssystems beinhalten.

7. Verfahren nach einem der vorangehenden Ansprüche, **dadurch gekennzeichnet**, daß während des Aufrufes der Zielfunktion durch die Startfunktion ein erneuter Aufruf der Zielfunktion durch eine Funktion außerhalb der Zielfunktion blockiert ist.

8. Verfahren nach einem der vorangehenden Ansprüche, **dadurch gekennzeichnet**, daß die die Zielfunktion enthaltende Task nur derart Zugriffe auf Daten ausführt, die von der Zielfunktion verwendet oder verändert werden können, daß die Zugriffe innerhalb eines Prozessortaktes abgeschlossen oder über Flaggen verriegelt sind.

9. Softwareprogramm zum Laden in den Arbeitsspeicher eines durch ein Betriebssystem betriebenen Computers mit einem Prozessor mit einer Memory-Management-Unit (MMU), **dadurch gekennzeichnet**, daß es eine Task mit einer Startfunktion zur Durchführung eines Verfahrens gemäß einem der vorangehenden Ansprüche umfaßt.

10. Softwareprogramm nach Anspruch 9, **dadurch gekennzeichnet**, daß es eine zweite Task mit einer Zielfunktion zur Durchführung eines Verfahrens gemäß einem der vorangehenden Ansprüche umfaßt.

11. Maschinenlesbarer Datenträger mit einem auf dem Datenträger abgespeicherten Softwareprogramm nach Anspruch 9 oder 10.

* * * * * * *

Zusammenfassung:

Die Erfindung betrifft ein Verfahren zum direkten Aufrufen einer Zielfunktion mittels einer Startfunktion durch einen Prozessor mit einer Memory-Management-Unit (MMU) in einem durch ein Betriebssystem betriebenen Computer.

Bei heutigen Multitasking-Betriebssystemen wird der Aufruf einer Funktion einer ersten Task durch eine zweite Task durch den Task-Scheduler des Betriebssystems ausgeführt und verwaltet. Der Zeitpunkt der Durchführung der aufgerufenen Funktion ist unbestimmt und von dem Betriebssystem sowie den im jeweiligen Zeitpunkt durch das Betriebssystem verwalteten Tasks abhängig.

Aufgabe der Erfindung ist es, ein Verfahren zu schaffen, welches einen zeitlich bestimmten Aufruf einer Funktion ermöglicht, der unmittelbar im Anschluß an den Aufruf ausgeführt wird.

Diese Aufgabe wird dadurch gelöst, daß die Startfunktion Bestandteil einer ersten Task mit einem ersten Speicherkontext ist und die Zielfunktion in einem anderen Speicherkontext liegt und daß die erste Task einen Kontextwechsel von dem ersten in den anderen Speicherkontext durchführt, der nach der Ausführung der Zielfunktion rückgängig gemacht wird.

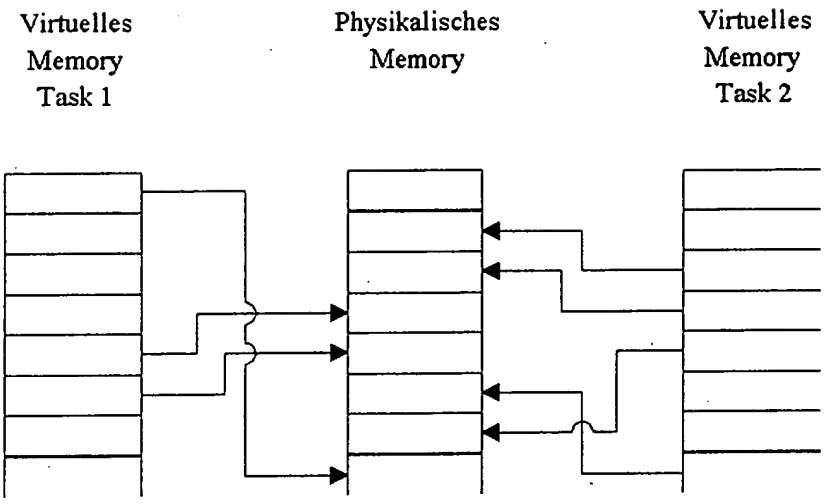
Virtuelles
Memory
Task 1
Physikalisches
Memory
Virtuelles
Memory
Task 2

FIG. 1

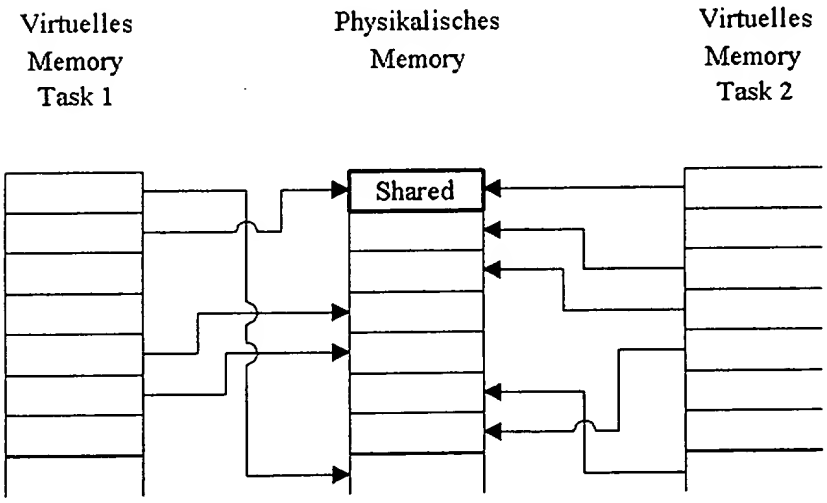
Virtuelles
Memory
Task 1
Physikalisches
Memory
Virtuelles
Memory
Task 2
Shared

FIG. 2

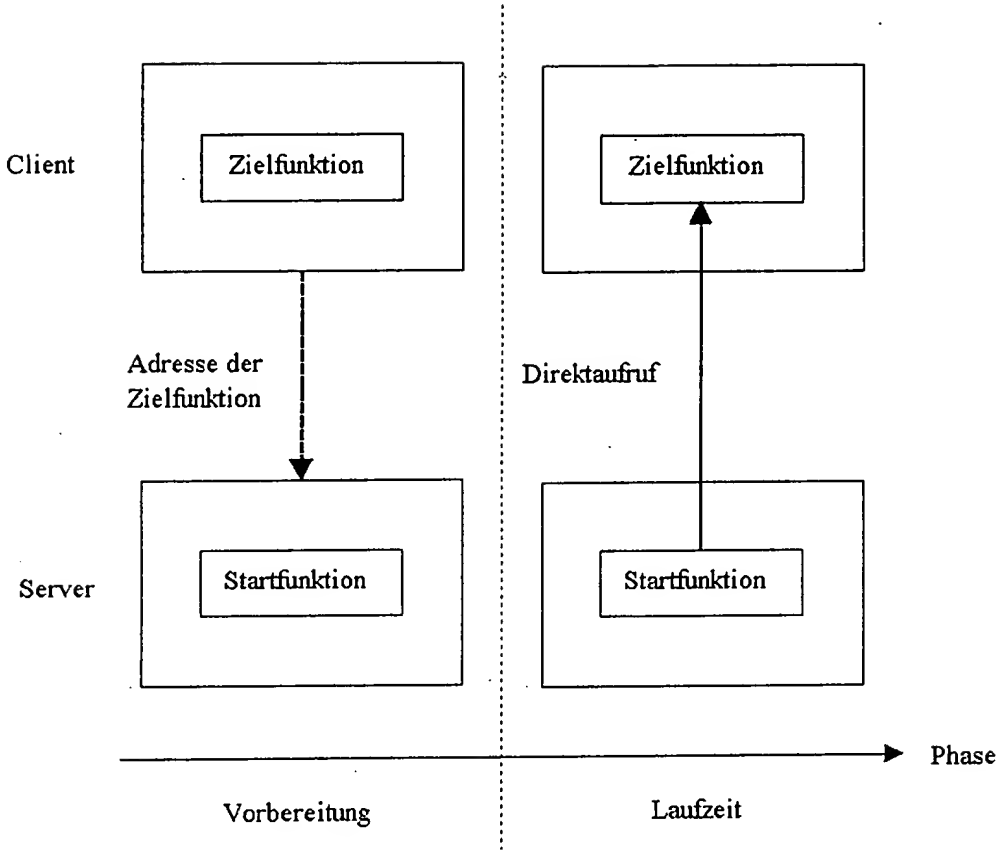
Client
Zielfunktion
Zielfunktion
Adresse der
Zielfunktion
Direktaufruf
Server
Startfunktion
Startfunktion
Phase
Vorbereitung
Laufzeit

FIG. 3